The stability problem for verification of concurrent object-oriented programs

Marieke Huisman, Clément Hurlin

The Stability Problem for Verification of Concurrent Object-Oriented Programs

Marieke Huisman[1,2]

INRIA Sophia Antipolis
2004, route des Lucioles BP 93
06902 Sophia Antipolis, France

Clément Hurlin[1,3]

INRIA Sophia Antipolis
2004, route des Lucioles BP 93
06902 Sophia Antipolis, France

Abstract

Modular static verification of concurrent object-oriented programs remains a challenge. This paper discusses the impact of concurrency on the use and meaning of behavioural specifications, and in particular on method contracts and class invariants.
Atomicity of methods is often advocated as a solution to the problem of verification of multithreaded programs. However, in a design-by-contract framework atomicity in itself is not sufficient, because it does not consider specifications. Instead, we propose to use the notion of stability of method contracts to allow sound modular reasoning about method calls. A contract is stable if it cannot be broken by interferences from concurrent threads. We explain why stability of contracts cannot always be shown directly, and we speculate about different approaches to prove stability. Finally, we outline how a proof obligation generator for sequential programs can be extended to one for concurrent programs by using stability information.
This paper does not present a full technical solution to the problem, but instead shows how it can be decomposed into several smaller subproblems. For each subproblem, a solution is sketched, but the technical details still need to be worked out.

Keywords: Multithreading, Design by Contract, stability.

1 Introduction

With the high demands on performance of software, the use of concurrency has become compulsory. Unfortunately, often the gains in speed are cancelled out by the bugs due to the use of concurrency. Therefore, formal techniques to analyse and reason about concurrent programs are necessary. Model checking provides a partial solution, but because of the state space explosion, this often does not scale

[1] This work is partially funded by the IST programme of the EC, under the IST-FET-2005-015905 Mobius project, and the French national research organisation (ANR), under the ANR-06-SETIN-010 ParSec project.

[2] Email: marieke.huisman@sophia.inria.fr

[3] Email: clement.hurlin@sophia.inria.fr

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

up to complex programs and properties. Instead, we propose to use logic-based techniques for the verification of concurrent programs. We focus in particular on the verification of multithreaded Java programs, where threads communicate via a shared global memory. In the literature two classical approaches exist to verify such programs: the non-modular Owicki-Gries method [17], and Jones's modular rely-guarantee method [9]. However, both approaches require one to write very detailed specifications about the interactions between the different threads, and therefore they do not scale.

Instead, we take a different approach, and identify as many code fragments as possible that can be verified sequentially. In particular, we use method and class specifications, as advocated in the Design by Contract approach [14] (and its Java-instantiation JML [11]), and discuss the impact of multithreading on their use and verification. The basic idea behind Design by Contract is that *preconditions* impose conditions on clients, while *postconditions* provide guarantees. In addition, *class invariants* specify properties that hold throughout the execution.

The use of such specifications provides the basis for sound modular verification of sequential object-oriented programs [15]: when verifying a code fragment containing a method call, the method call can be abstracted by its specification. Consider for example the following code fragment, annotated with JML.

```
//@ requires P;
//@ ensures Q;
void call(){ ... }

void method(MyObject o){
  o.call();
  // (1) }
```

When reasoning sequentially, one can assume that `Q[o/this]` holds after `o.call()` (at `(1)`). However, in a multithreaded setting this need not be the case: any other thread simultaneously executing within the object pointed to by `o` can change the validity of `Q[o/this]`, and in particular, `Q[o/this]` might be invalidated between the end of `call` and the continuation of `method`. Therefore we propose to use the notion of *stable* contract (cf. [3]), to indicate that a contract cannot be invalidated by another thread. Contract stability is crucial for the verification of multithreaded programs: after a method call, a stable postcondition can be used as a precondition for the next statement, whereas an unstable postcondition must be discarded. Similarly, a method implementation can only rely on a stable precondition.

This paper discusses how contract stability can be used to reason about multithreaded programs, and it sketches several speculative ideas how it can be proven. Section 2 discusses in more detail the difficulties of reasoning with method specifications for multithreaded programs. Section 3 defines stability of contracts and discusses how locking, confinement, immutability, and semantics might be used to prove stability. Section 4 demonstrates how to extend a sequential verification condition generator for multithreading, while Section 5 concludes.

2 Reasoning with Specifications in Concurrent Programs

2.1 Method Specifications

Traditionally, *atomicity* [5] has been advocated as a means to decide whether a method could be verified sequentially. A method is said to be atomic if it contains at most one instruction that is sensitive to interference (in the special case where the method contains no such instruction, it is said to be *independent*). If a method is atomic, its method body can be verified without considering interleavings from other threads. However, existing atomicity analyses do not take method specifications into account, and in particular they do not consider whether a pre- or postcondition can be invalidated by another thread. Therefore, it is not possible to reason *about* calls to atomic methods in terms of method specifications. In particular, if an atomic method has an unstable contract, reasoning about calls to this method cannot be done in the standard modular way. However, contract stability is not strictly a stronger notion than atomicity: if the instructions that break the atomicity of the method do not influence validity of the stable method specification, it still can be verified sequentially if the method respects its contract.

Also, we would like to remark that atomicity (and independence) are often conditional properties, i.e., they only hold under particular conditions (see also [5]). Typical examples of such conditions are that a particular lock is held, or that a method parameter is local to a thread. However, when reasoning in terms of method specifications, we think it is sufficient to list these conditions as part of the method's precondition. An alternative approach would be to list such conditions separately, and to interpret the method specification differently, depending on whether this condition is satisfied or not. If the condition is satisfied, and the contract is stable, it is interpreted as is. However, if the condition is not satisfied, all unstable parts should be removed from the contract (cf. [3]). But this would mean that from the client's point of view, methods could have different behaviours. We think this is an undesirable and counter-intuitive situation; making these conditions part of the precondition ensures that the method only can be called in cases where the contract is stable, and thus where one can rely on the method behaving as specified.

Finally, notice that also the meaning of the `modifies` clause must be changed. This clause specifies which variables may be modified during a method call, and all others are implicitly unchanged. However, in a concurrent setting unstable variables can always be modified by another thread. Therefore, a `modifies` clause only implicitly specifies which *stable* variables remain unchanged.

2.2 Class specifications: Invariants and Constraints

The standard meaning of invariants also needs to be revised in a multithreaded context. JML defines a visible-state semantics for class invariants: *all* invariants of *all* allocated objects have to hold in *all* visible states, i.e., basically all states in which a method is called or finished (except for so-called "helper" methods) [12, §8.2]. This means in particular that invariants may be freely broken *inside* a method body.

But in a multithreaded program, when one thread is inside a method body, and thus has the right to break the invariants, another thread might just be entering a method, and thus require the invariants to hold. This is particularly relevant for *true concurrent objects* [19], where several threads may access simultaneously the same object. However, even if we would exclude such behaviour (which could decrease performance [1]), the problem remains, because JML requires *all* invariants to hold in a visible state. Thus, even if different parallel threads would never be allowed to access the same part of memory at the same time, the standard JML semantics would have to be adapted, by requiring for example that only the invariants that are related to that part of memory that can be accessed by the current thread have to hold. However, deciding statically which invariants are relevant to a particular program point is an open problem.

A partial solution is the introduction of so-called *strong invariants*, i.e., invariants that are never broken, not even temporarily. Another possibility is to specify explicitly the properties on which a thread can rely when it has certain locks. This could be expressed using an expression like `locking_rely(l1, ..., ln) P`, meaning that if a thread acquires the locks `l1, ..., ln`, it can assume the property `P`, and when it releases the locks, it has to establish `P`. Alternatively stated, the environment guarantees `P` when `l1, ..., ln` are held. One can also imagine further combinations of the different possibilities, where the thread holding the locks has to ensure that the predicate `P` holds in any of its visible states. A particular instance of this would be to restrict object invariants to specify properties about the state that is protected by the object's lock. Thus, having the object's lock means that one can rely on the object's invariants.

A related problem exists for constraints. A constraint describes a relation that is supposed to hold between every pair of consecutive visible states. But in a multithreaded setting these might belong to different threads. Thus, a naive approach to verify a constraint would be to consider all possible interleavings of the different threads, but this results in non-modular verification. Another possibility is to redefine the notion of constraint, so that it only relates visible states within the same thread. However, in that case, one needs to ensure that other threads cannot break the constraint in-between related states.

In the rest of this paper, we will focus on how we can show that method specifications are stable, and how this can exploited for verification. We assume that the contracts have been extended with conditions arising from class invariants and history constraints. However, from the above, it should be clear that simply desugaring *all* class invariants and constraints into method pre- and postconditions would make it virtually impossible to show that such an extended method contract is stable. Therefore, in addition we will need to develop the means to modularise class invariants and constraints over the program - so that the method contract is extended only with the relevant class specifications.

3 Stability of Contracts

The stability of a contract depends on how variables that have to be read to evaluate the contract can be modified by threads. We use the term *footprint* to denote

an upper bound on the set of locations that are accessed during contract evaluation. It depends on the expressiveness of the specification language how precisely a contract's footprint can be computed. For a contract which only reads fields and contains no qualified expressions, the contract's footprint is the set of all fields appearing in the contract.

However, for more complex specification expressions, computing the footprint is more complicated. For example, to compute the footprint of a contract containing a quantifier, we need to be able to determine the quantifier's domain. Fortunately, often the domain of a quantifier is finite, e.g., a universal quantifier ranging over all elements of an array. As an example, the footprint of `oneify`'s contract below is the whole array `a`; thus if no elements of `a` can be written by other threads, `oneify`'s contract is stable.

```
//@ ensures (\forall int i; i >= 0 && i < a.length; a[i] == 1);
void oneify(int[] a){ for (int i = 0; i < a.length; i++) a[i] = 1; }
```

Another problem is when the footprint of a contract cannot be expressed as a list of accessed fields. For example, the footprint of `multiply`'s contract below depends on the way `Node` objects are linked. To show stability of `multiply`'s contract, one has to show that all objects that are recursively accessible via the `next` field are stable (i.e., cannot be modified by concurrent threads). We plan to investigate whether we can use JML's ownership type system [4], to show stability of contracts in such cases.

```
class Node{
  int x; Node next;
  //@ invariant x > 0 && ((next != null) ==> next.x < this.x);

  //@ measured_by(this.x);
  //@ ensures (next != null) ==> (\result == x * next.multiply());
  //@ ensures (next == null) ==> (\result == x);
  /*@ pure @*/ int multiply(){
   if(next == null)
     return x;
   else
     return x * next.multiply(); }}
```

To summarize, the footprint of a contract indicates which objects should be thread-local or lock-protected so that the contract is stable. In some simple cases, a contract's footprint can be computed directly, but in general this is not feasible. The next paragraph describes a permission system used to control concurrent access to objects. This permission system permits to show properties that can be used to prove contract stability. The remainder of this section then shows how stability is proven.

3.1 A Program in Need of Permissions

In order to show stability of contracts, we use a permission system whose general ideas are described elsewhere [7]. This system is a generalization of Boyland's

fractional permissions system [2] to object-oriented programs with dynamic thread creation, joins and locks. Our permission system permits to verify properties that are sufficient to show stability. In the following we recall the most relevant features of our system.

A class can be annotated with the keyword `permission` to indicate the permissions that exist for each instance of this class. Permissions can be base permissions, object permissions, or lock permissions. Base permissions have the form `f : \split(n)` where `f` is a field of the class considered and `n` is a natural number. Intuitively, this means there are at most 2^n threads accessing the field at the same time (since $2^0 = 1$, `\split(0)` means only one thread has access). Henceforth, permission `f : \split(0)` allows to write to and read to field `f`, permission `f : \split(n)` (with `n`> 0) allows solely to read. A permission can be *split* into two smaller permissions using the equivalence `\split(n)` $\equiv$ `\split(n+1) && \split(n+1)`. To alleviate the annotation burden we use `W` as a shorthand for `\split(0)` and `R` as a shorthand for `\split(1)`.

Object permissions have the form `f : W (\split(p,n))` or `f : R (\split(p,n))` where `p` is a permission and `n` is a natural number. These permissions contain the base permission `f : W` (or `f : R`) and permission `\split(p,n)` on the object pointed to by `f`. As with base permissions, we have the equivalence `\split(p,n)` $\equiv$ `\split(p,n+1) && \split(p,n+1)`. When `n` is 0 we simply write `p`.

Lock permissions have the form `\split({l},n)` where `l` is a non-primitive final field of the class considered and `n` a natural number. For any `n`, permission `\split({l},n)` gives the right to lock `l`. When `n` is 0 we simply write `{l}`. A lock permission comes with a lock clause written `lock {l} = p, ..., q` to indicate that permissions `p, ..., q` are obtained when `l` is locked.

For every field `f` of an object, the permission system guarantees that *(i)* only one permission allows to write to `f`, and *(ii)* if a read permission to `f` exists, then there is no write permission to `f`. These conditions are crucial to show stability. Finally, our system is defined as an extension of JML [11], so permissions requirements and guarantees are expressed as part of method contracts.

The rest of this section speculates about different possibilities to prove contract stability by using our permission system.

3.2 Stability by Locking

Most concurrent programs rely on locking to achieve correct synchronization. Locking can be used to provide exclusive access to parts of the heap. Existing methodologies have ways to protect individual fields [18,7] or whole classes [8] by locks. The permission system described before allows to specify locking policies. Adherence to the specified locking policy of a class helps to show contract stability of the class's methods. Note that – although we do not detail it here – aliasing complicates stability checking: determining which lock protects which object can require aliasing information.

Class `Point` illustrates how locking policies help to show stability. Class `Point`'s `permissions` clause define that there exists a permission `p` per `Point` object. Per-

mission p allows to synchronise on the Point object considered. Further, the lock clause indicates that synchronising on a Point object gives write permission to fields x and y of this object.

Precondition of method shiftX's requires callers to have permission {this} i.e., to synchronise on this before calling shiftX. This behavior is called *client-side locking* [8]. Note that because permission p means the right to lock this, it cannot be written as a precondition for shiftX whose correctness relies on this being locked before calling (which is the meaning of requires {this}).

```
class Point{
  int x, y;

  //@ permission p = {this};
  //@ lock {this} = x : W, y : W;

  //@ requires {this};
  //@ ensures  {this} && x == \old(x) + delta;
  void shiftX(int delta){ x += delta; }}
```

Client-side locking rules out interferences from other threads, thus it permits to show stability. Since shiftX's caller holds the lock on this, and since writes and reads to x are only allowed when this is held (because of the lock clause), throughout shiftX's execution concurrent accesses to x are not possible. Thus, all locations in shiftX's footprint ({\old(x),x}) cannot be written by other threads and the contract is stable.

3.3 Stability by Confinement

Another technique to ensure stability is to control concurrent access to objects. This allows one for example to show that an object can only be accessed by a single thread (it is said to be *thread local*). This technique is also particularly useful for lock-free algorithms (like the ones described in [13]), where accesses to objects are distributed by the algorithm. Our permission system supports such programs.

For example, class DoubleInt below specifies that there exists two permissions p and q on instances of class DoubleInt. Permission p allows to write to and read from field x (similarly for permission q and field y). Method incX of class DoubleInt requires permission p on this and returns the same permission when the call returns.

```
class DoubleInt{
  int x, y;

  //@ permission p = x : W;
  //@            q = y : W;

  //@ ensures p && q && x == 0 && y == 0;
  public DoubleInt(){ x = 0; y = 0; }
```

```
  //@ requires p && x == 0;
  //@ ensures  p && x == 1;
  public void incX(){ x++; }

  //@ requires q && y == 0;
  //@ ensures  q && y == 1;
  public void incY(){ y++; }}
```

The notation `r = di : R (q)` in class `IncMachine` below specifies an object permission: it contains a base permission `di : R` and also permission `q` on the `DoubleInt` object pointed to by `di`. This shows how permissions can be encapsulated into other permissions (following the object-oriented paradigm).

In method `main` below, the main thread first creates the `DoubleInt` object `di` and obtains `di`'s permissions `p` and `q`. Then, the main thread creates the `IncMachine` thread `t` and encapsulates `di`'s permission `q` in `t`'s permission `r`. When `t` is started, permission `r` is transferred from the main thread to `t`. Then, the `main` thread can call `di.incX()` (using `di`'s permission `p`), while the `IncMachine` thread can execute `di.incY()` (using its permission `r` which contains `di`'s permission `q`). This makes class `DoubleInt` truly concurrent, because two threads can execute simultaneously within an instance of this class. Also note that the main thread cannot call `di.incY()` once it has created `t`. Permissions are split when the `main` thread creates `t`: the main thread keeps `di`'s permission `p` but `di`'s permission `q` is encapsulated into `t`'s permission `r`, therefore becoming inaccessible to the main thread.

```
class IncMachine extends Thread{
  final DoubleInt di;

  //@ permission r = di : R (q);

  //@ requires di.q;
  //@ ensures  r;
  public IncMachine(DoubleInt di){ this.di = di; }

  //@ requires r;
  //@ ensures  r;
  public void run(){ di.incY(); }

    public static void main(){      // permissions owned by threads
    DoubleInt di = new DoubleInt(); // main has di.q and di.p
    Thread t = new IncMachine(di);  // main has di.p and t.r

    t.start();                      // main has di.p, t has r
    di.incX();
    // (1)
    }}
```

Because there is only one permission `p` to write to field `x` of any instance of class `DoubleInt`, concurrent writes to `di`'s field `x` are impossible and therefore `incX`'s

postconditions is stable: `di.x == 1` can be assumed at point `(1)`. Thus, the permission system allows us to show stability of contracts of classes designed to be accessed by a single writer thread without locking: this is particularly useful for lock-free programs.

3.4 Stability by Immutability

Another way to prove stability of contracts is to use the notion of immutability [6]. An object is said to be *immutable* if it is never written after its initialization. Therefore, it is safe to access it without synchronization. With the annotation system described above, immutability can be expressed by `R` base permissions. Immutability can be used to show stability of contracts. For example, class `Fraction` below (adapted from Lea [10]) is an immutable class. Clause `permission` of class `Fraction` specifies a permission `p` which allows solely to read fields `n` and `d` of `Fraction` objects.

In order for multiple threads to simultaneously access `Fraction` objects, we need to distribute permission `p` among different threads. Our system supports this by splitting permissions. We use `\part(p)` to denote a *part* of permission `p`, that is `p` or `p` split any number of times (`\part(p)` is desugared into `(\exists n. split(p,n) && n >= 0)`).

Precondition of method `plus` requires callers to have a part of `p`. Stability of `plus`'s contract is trivial to prove, because its footprint only contains accesses to readonly fields (and the permission system ensures that there cannot be - interfering - write permissions to these fields in other parts of the program).

```
class Fraction{
  final int n; // numerator
  final int d; // denominator

  //@ permission p = n : R, d : R;

  //@ ensures p;
  public Fraction(int num, int den){ n = num; d = den; }

  //@ requires \part(p) && \part(f.p);
  //@ ensures  \part(p) && \part(f.p) && \result.p &&
  //@          \result.n == n * f.d + f.n * d && \result.d == d * f.d;
  public Fraction plus(Fraction f){
    return new Fraction(n * f.d + f.n * d, d * f.d); }}
```

3.5 Stability by Semantics

The techniques to show stability described above are syntactical techniques. In the following, we sketch an example to give the reader an intuitive notion of stability by semantics. Class `Account` below (adapted from [8]) uses a JML constraint (a simple temporal property) to express that items in the history are never deleted. Because the semantics of constraints have been influenced by rely-guarantee techniques, they can be used to show stability of contracts in a manner reminiscent of rely-guarantee

reasoning (but applied to object-oriented programs).

```
class Account{
  //@ constraint (\forall Integer x;
  //@               \old(history).contains(x); history.contains(x));

  final Vector<Integer> history = new Vector<Integer>();
  int balance = 0;

  //@ permission p = {this};
  //@ lock {this} = balance : W, history : ...;

  //@ requires \part(p);
  //@ ensures  \part(p) &&
  //@          (\exists Integer x; history.contains(x); x == amount);
  synchronized void deposit(Integer amount){
    balance+=amount;
    history.add(amount); }}
```

In this example, `deposit`'s postcondition is sensible to interferences from other threads: between `deposit`'s returning and the caller resuming a thread may call `deposit`, thus adding an item to the history. However, the constraint and the fact that `amount` is put in the history during `deposit`'s executions entail stability of `deposit`'s postcondition. Generally, proving stability by semantics consists in using constraints to give additional assumptions in the presence of interferences from concurrent threads.

4 Lifting Sequential Program Verification to Concurrency

Above, we have shown how method's contracts can be shown to be stable, so that they can be used for reasoning about method calls. However, we also need a technique to discard properties that we no longer can rely upon. In particular, whenever the stability of objects changes, because of sharing or releasing locks, the unstable expressions need to be discarded from the intermediate assertions. This technique is inspired by the "havoc" approach introduced in the Boogie methodology [8]. However the stability information obtained is more fine-grained than the approach cited: less havoc statements are generated resulting in easier proof obligations. In particular, the Boogie approach forces programmers to protect whole classes by locks whereas we allow a per field protection. This permits us to havoc only certain fields of objects while the Boogie approach always havoc all fields.

We illustrate the discarding mechanism by an example, showing how the sequential verification of a `CommonWarehouse` is lifted to a concurrent one. The example uses a Floyd-Hoare-like proof outline, but a similar technique can be used to lift proof obligation generators based on, e.g., weakest precondition or strongest postcondition calculi from sequential to concurrent program verification.

Class `Quantity` in Figure 1 defines a permission `p` that gives write access to field

```
class Quantity{
  volatile int i; //@ invariant i >= 0;

  //@ permission p = i : W;

  //@ requires initial >= 0;
  //@ ensures p && i == initial;
  public Quantity(int initial){ i = initial; }

  //@ requires p && plus >= 0;
  //@ modifies i;
  //@ ensures p && i >= plus;
  public void add(int plus){ i += plus; }}

class CommonWarehouse{
  //@ permission q = ...;

  //@ requires \part(q) && o.p;
  //@ ensures  \part(q);
  void lend(Quantity o){ ... }

  //@ requires \part(q);
  //@ ensures  \part(q) && o.p;
  void takeBack(Quantity o){ ... }

  //@ requires \part(c.q) && j >= 0 && k >= 0;
  //@ ensures  \part(c.q) && \result.p && \result.i >= k;
  public static Quantity main(CommonWarehouse c, int j, int k){
    Quantity o = new Quantity(j);
      [o.p]    {o.i = j}_s ⇝ {o.i = j}_c
    c.lend(o);
      []       {o.i = j}_s ⇝ {⊤}_c
    c.takeBack(o);
      [o.p]    {o.i = j}_s ⇝ {⊤}_c
    o.add(k);
      [o.p]    {o.i ≥ k}_s ⇝ {o.i ≥ k}_c
   return o; }}
```

Fig. 1. Example: from sequential proof outline to concurrent proof outline

`i`. Note that class `Quantity`'s contracts are stable by confinement. Notice further that its invariant is a strong invariant. The methods of class `CommonWarehouse` are annotated with pre- and postconditions indicating how permissions of parameters are updated. For example, `lend`'s callers must have a part of permission `q` on the receiver and permission `p` on parameter `o`.

Method `main` is annotated with a proof outline (only showing relevant formulas). First, between square brackets, the evolution of permissions owned by the main

thread is shown. Initially, the main thread has exclusive access to the new `Quantity` object, because it has permission `p` on this object (and as `Quantity`'s `permissions` clause shows, only one such permission may exist on a `Quantity` object). After calling `lend`, it gives away this permission (as indicated by `lend`'s contract), and the object pointed to by `o` becomes unstable, as it can be accessed by other threads. After calling `takeBack`, the main thread regains exclusive access to the `Quantity` object (because it gains back `o`'s permission `p`), and thus it knows `o.i` cannot be written by concurrent threads. The second column (enclosed with $\{\}_s$) are the *sequential formulas* obtained by a strongest postcondition calculation. Finally, the third column (enclosed with $\{\}_c$) shows the *concurrent formulas*, the result of weakening the intermediate assertions based on the stability information.

After constructing the new `Quantity` object and assigning it to `o`, sequential formula `o.i = j` holds, because of the constructor's postcondition. As this postcondition is stable, it is also a concurrent formula. Because of the implicit `modifies \nothing` clause, `lend` and `takeBack` do not change `o`'s contents, and the sequential formula remains `o.i = j`. However, the call to `lend` makes `o` unstable: another thread gains `o`'s permission `p` and this thread may write to `o.i`, therefore properties about `o` need to be discarded in the concurrent formula. After the call to `takeBack`, `main` has exclusive access to `o` again, but as we do not know what other threads did with the object in the mean time, we cannot assume anything about `o.i` here. Only after the call to `add`, we know something about `o.i`'s value. This information also holds in a concurrent setting, because `o` is stable: `o.i` cannot be written by other threads. However, notice that if the postcondition of `add` had contained the old value of `i`, the concurrent formula would have reduced to $\top$ again.

This shows how the stability information provided by the permission system can be used to compute intermediate assertions for concurrent programs. This approach is modular because *(i)* concurrency aspects are delegated to the permission system, *(ii)* interferences of concurrency with the intermediate assertions occur only at weakening points, and *(iii)* program behaviour is abstracted by method specifications. This improves for example over rely-guarantee techniques, where the stability information flows through the proof.

5 Conclusions and Future Work

This paper sketches a modular verification technique, based on method specifications for multithreaded Java programs. The main idea is that method specifications should be stable, i.e., their validity should not be affected by other threads. If method specifications are stable, modular sequential verification techniques can be adapted for a concurrent setting. We show how locking, controlling object access and immutability can be used to show stability of contracts. We also show how stability information allows to lift a sequential proof outline to a concurrent proof outline.

This paper does not describe finished work; instead it is a first step towards the development of a verification technique for multithreaded programs, without putting any major restrictions on the programming model used.

For this technique to be fully operational, the following topics need to be ad-

dressed: *(i)* class invariants and history constraints must be handled appropriately, probably requiring that certain locks are held before such specifications can be assumed, *(ii)* more precise techniques to compute contract's footprints have to be developed, *(iii)* techniques for stability checking need to be completed and implemented, and *(iv)* we need to extend a verification condition generator for sequential programs to concurrent programs, based on the weakening procedure described in Section 4.

Related Work

Jacobs et al. [8] recommend *client-side locking* to force contracts to rely on stable objects and require contracts to perform only legal access. They do not discuss how to check this for complex specifications. They use an ownership system to control object accesses, however in a more restrictive way, as they exclude for example, true concurrency. Contrary to Jacobs et al., we try not to impose a programming model and aim at verifying more varied patterns.

Rodriguez et al. [18] use the term *internal interference* to denote that a thread may change data observable by a method executed by another thread. They rule out internal interferences by using atomicity and independence i.e., by showing that an interleaved execution of a method can always be reduced to a sequential execution. They call *external interference* the problem of a thread making observable changes between a method call of another thread and the method's entry of this thread (or between the method's exit and the thread's resuming). As we pointed out in section 2, atomicity and independence are not sufficient to avoid this problem and the method presented in this work suffers from this defect: it is advocated that contracts should solely rely on *thread safe* objects, i.e., local or locked objects, but it is not described how this is enforced. Our technique handles both kind of interferences in a sound way.

Nienaltowski and Meyer [16] address the stability problem by having an alternative semantics for contracts. Preconditions are treated as wait-conditions: a non-satisfied precondition forces the client to wait until it becomes true. Postconditions are projected into the future: a postcondition on an object is required to be true only when this object is accessed. However, as the authors point out this solution raises liveness issues. Furthermore, it cannot be applied to Java where method calls and returns do not respect this alternative semantics.

Calcagno et al. [3] address the stability problem by *stabilizing* assertions: from unstable properties weaker stable properties are computed, using a fixpoint computation. This approach is not integrated into a Design by Contract framework. Notice that our technique does not require stabilization, we simply impose contracts to be stable. This can be seen as a worst case stabilization: we consider that shared objects can be affected in any way by other threads.

Acknowledgements

We thank our partners in the Mobius project, in particular Christian Haack, Erik Poll for their useful and constructive comments on the stability issue. We also thank Gustavo Petri for his insightful remarks.

References

[1] C. Baquero, R. Oliveira, and F. Moura. Integration of concurrency control in a language with subtyping and subclassing. In *USENIX Conference on Object-Oriented Technologies*, Monterey, California, 1995.

[2] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer-Verlag, 2003.

[3] C. Calcagno, M. Parkinson, and V. Vafeidis. Modular safety checking for fine-grained concurrency, 2007. Submitted.

[4] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, October 2005.

[5] C. Flanagan and S. Qadeer. Types for atomicity. In *Types in Language Design and Implementation*. Association of Computing Machinery Press, 2003.

[6] C. Haack, E. Poll, J. Schäfer, and A. Schubert. Immutable objects for a Java-like language. In R. De Nicola, editor, *European Symposium on Programming*, volume 4421 of *LNCS*, pages 347–362. Springer-Verlag, 2007.

[7] M. Huisman and C. Hurlin. Thread capability annotations for common multithreaded programming patterns, 2007. Manuscript, http://www-sop.inria.fr/everest/Clement.Hurlin/publis/annotations-patterns.pdf.

[8] B. Jacobs, K.R.M. Leino, F. Piessens, and W. Schulte. Safe concurrency for aggregate objects with invariants. In *Software Engineering and Formal Methods*, Koblenz, Germany, 2005.

[9] C.B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[10] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns (Second Edition)*. Addison-Wesley Publishing Company, Boston, MA, USA, 1999.

[11] G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06y, Iowa State University, 1998. (revised since then 2004).

[12] G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. *JML Reference Manual*, July 2005. In Progress. Department of Computer Science, Iowa State University. Available from http://www.jmlspecs.org.

[13] C.E. Leiserson and H. Prokop. Minicourse on multithreaded programming, July 1998. http://supertech.csail.mit.edu/cilk/papers/index.html.

[14] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[15] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[16] P. Nienaltowski and B. Meyer. Contracts for concurrency. In *Symposium on Concurrency, Real-Time, and Distribution in Eiffel-like Languages*, York, United Kingdom, July 2006.

[17] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica Journal*, 6:319–340, 1975.

[18] E. Rodríguez, M.B. Dwyer, C. Flanagan, J. Hatcliff, G.T. Leavens, and Robby. Extending JML for modular specification and verification of multi-threaded programs. In A.P. Black, editor, *European Conference on Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 551–576. Springer-Verlag, July 2005.

[19] L. Thomas. Inheritance anomaly in true concurrent object oriented languages: A proposal. In *TENCON*, pages 541–545. IEEE Press, August 1994.